Data synchronized pipeline architecture: Pipelining in multiprocessor environments

Yvon Jégou, André Seznec

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 503

DATA SYNCHRONIZED PIPELINE ARCHITECTURE : PIPELINING IN MULTIPROCESSOR ENVIRONMENTS

Yvon JEGOU
André SEZNEC

Mars 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n° 283
Janvier 1986 - 36 pages

Data Synchronized Pipeline Architecture :

pipelining in multiprocessor environments

Yvon JEGOU
André SEZNEC
IRISA / INRIA
Campus de Beaulieu
35042 RENNES CEDEX
FRANCE

ABSTRACT : To satisfy the growing needs for computing power, a high degree of parallelism will be necessary in future supercomputers. Up to the late 70s, supercomputers where either multiprocessors (SIMD-MIMD) or pipelined monoprocessors. Future industrial realizations should combine these two levels of parallelism. In a multiprocessor, classical pipeline controls become inefficient because the interdependant behaviors of the processing elements cannot be foreseen neither at compile time nor at decode time. In this paper, we introduce a new model of pipeline architecture : the Data Synchronized Pipeline Architecture (DSPA). Based on an independant sequencing of the functional units, this model allows a high degree of parallelism in the pipeline, even in the case of unforseeable behaviors of some ressource.

RESUME : Les besoins en puissance de calcul ne pourront pas être satisfaits sans un haut niveau de parallélisme dans les futurs supercalculateurs. Jusqu'à la fin des années 70, les supercalculateurs étaient soit des multiprocesseurs, soit des monoprocesseurs pipelines. Les réalisations futures devraient combiner ces deux niveaux de parallélisme. Les techniques classiques de contrôle des pipelines sont inadaptées pour les multiprocesseurs car les comportements des processeurs élémentaires ne peuvent être prévus ni à la compilation, ni même au décodage. Dans ce papier, nous définissons un nouveau modèle d'architecture pipeline : le Data Synchronized Pipeline Architecture (DSPA). Basé sur un séquencement indépendant des unités fonctionnelles, ce modèle permet un haut niveau de parallélisme, dans un processeur pipeline, même dans le cas où le comportement de certaines ressources est imprévisible.

I Introduction

Needs for computing power seem unlimited in various scientific applications. During the last ten years, tremendous progresses have been done in the domain of componant integration. But todays' supercomputer clocks are of the same order as those of ten years ago supercomputers. E.g. the clock of the Cray2 (1985) is only three times faster than the one of the Cray1 (1976).

Requirements for performances have lead manufacturers to the design of parallel structures. The first industrial parallel supercomputers were pipeline processors (Cray1, CDC Cyber 205, ..). Up today, these pipeline computers can be considered as the state of art in monoprocessor architecture. Since the late 1970's, a lot of multiprocessor projects have been initiated [Ga83][Go83][Si81][Ho85]. In future industrial realizations of these ambitious projects, the elementary processors will be pipeline processors. A great attention must be taken in the design of these pipeline processors.

On a pipeline computer, the execution of an instruction stream generates concurrent activities on several functional units (FUs). These FUs may be pipelined. Even when the FUs are not pipelined, the successive FUs crossed by the data form a macropipeline. Performances of the computer depend heavily on the overlapping of successive instructions and are bounded by the throughput of the instruction decoder. In the case of vector instructions, simple pipeline control is possible because of the

regularity of the data and instruction streams. Optimizations of the code can be done at compile time. For example, performances of the Crayl can be considered as near optimum on classical vector instructions. That is the reason why pipeline computers are generally refered as vector processors. Unfortunately, performances of existent pipeline computers on scalar instructions are not as good as one can expect. The overlapping of successive instructions cannot always be studied at compile time because memory access conflicts or hazards may arise at execution time. Because in a multiprocessor architecture some hardware is shared, the behaviors of processing elements are interdependant. Moreover if decode time for vector instructions is small regardless to the occupation of the FUs, very fast algorithms have to be used for pipeline control in scalar mode, because the decoding delays become critical.

In section II, we recall the problems which exist on some classical pipeline computers. Then in the following sections, we present a new model of pipeline architecture : the Data Synchronized Pipeline Architecture (DSPA). This model has been developped in order to improve the performances of pipeline computers on scalar code. We also point out that using this model, a distributed decode of instructions allows very fast sequencing and efficient overlap for both vector and scalar instructions even in the cases where memory conflicts or hazards may occur.

II Existent pipeline models

1 Outlines

Today, a pipeline processor can be used as a monoprocessor (Cray1, FPS164, Cray XMP-1, Fujitsu VP 200,..) or as an elementary processor in a multiprocessor system (Cray XMP-4, Cray2, BSP,..). The BSP [Ku82] is a SIMD computer; its elementary processors (PEs) are pipelined. A set of vector instructions has been defined by the designers. These instructions are memory to memory instructions (operands are read on memory and the result is stored in memory). When the machine has been defined, the behavior of these instructions was studied and optimized (by introduction of delays between two functional units for example); reservation tables [Ba80][Ko81][Hw84] are stored in the machine. The SIMD structure of BSP allowed global pipeline control for all PEs. In general cases, the distinct FUs have not the same number of stages and many cases of memory conflicts may occur; even with a restricted number of vector instructions, the number of cases to be studied would increase exponentially with the number of operands involved by the instruction. In the BSP the number of cases to be studied remains quite limited; all the FUs have the same number of stages (the reservation table does not depend on the FUs but only on the number and the distribution of distinct FUs concerned with an instruction), and memory conflicts are almost avoided by a constant increment definition of a vector and the choice of a prime number of memory banks [La75][La82]. On the BSP, a good overlapping of distinct iterations of the same vector

instruction is performed; but there is no overlapping of the end of a vector instruction by the beginning of an other instruction : pipeline must be filled at the beginning of an instruction, and emptied at the end. Memory to memory instructions may be quite efficient on vector operands, but scalar instructions overlapping has to be done with other techniques while the restricted choice of vector definition increases the ratio of scalar code (e.g. loop 3 proposed later must be executed in scalar mode).

In today's multiprocessors, the behavior of a shared memory cannot be foreseen neither at compile time, nor at execution or decode time : it depends on the different processes being executed by the different elementary processors. Even when vector accesses are synchronized, memory conflicts may occur at anytime; memory behavior depends on the relations in the distribution of the vector elements, on the relations between two successive vector accesses and on the treatment of RAW (Read After Write) hazards [Ra77]: memory to memory architecture must be avoided when high speed scalar execution and asynchronous PE controls are expected.

We present the problems that remains on two existent pipelined monoprocessors.

2 Vector register machines

In register machines, the operands of an instruction are loaded in the functional units (FUs) from registers (except for loading a datum from memory in a register) and the results are stored in registers (except for storing a datum in memory). In

some pipeline computers, such as Crayl [Cr79], there are vector registers; a vector register may contain up to N words (in Crayl, N=64). A single instruction with vector register operands may generate up to N times the same operation on distinct data, results are then stored in a vector register. This approach seems very efficient on pure vector instructions.

Example 1:

```
      DO 1 I=1,N
    1 A(I)=B(I)+C(I)*D(I)
```

Only seven instructions are necessary to code the body of this loop for the Crayl.

```
Vector load C --> R1
Vector load D --> R2
R3 <-- R1*R2
Vector load B --> R4
R5 <-- R3 + R4
Vector store R5 --> A
Conditionnal jump
```

Time to decode such a loop is not critical : up to 256 accesses to memory are generated by these seven instructions and the Crayl can decode one instruction by cycle.

Unfortunately accesses to memory on the Crayl are performed in the order of the decode sequence; moreover no advance is taken on the decode sequence : the next instruction is decoded only when the present instruction is initiated. An instruction is

initiated only when the presences of all its operands are guaranteed; as there is no hardware mean to verify if the i^{th} element of a vector is present or no, to initiate a vector instruction (different from a load) the following condition must be guaranteed:

For every i, i cycles later the i^{th} element will be present.

These restrictions explain why 353 cycles are necessary to execute an iteration of loop 1 while the memory is only busied during 256 cycles. They also induce the same definition of a vector as in the BSP. The loop 2 is then treated in scalar mode :

Example 2:

```
      DO 2 I=1,N
    2 A(I)= B(H(I))+ C(I)
```

Cray1 is about ten times slower than the Fujitsu VP200 [Ta85] for the execution of this loop; because vector registers of addresses can be computed, the Fujitsu VP200 executes this loop as a vector loop. But loop 3 is executed in scalar mode on both machines:

Example 3:

```
      DO 3 I=1,N
    3 A(I)= A(H(I))+ B(I)*C(I)
```

Possible hazards may occur on this loop; as H(2) may be equal to 1, A(1) has to be written before A(H(2)) is read (at least the hardware should avoid reading A(H(2)) before writing A(1) if H(2)=1). The loop must be coded in scalar mode.

In scientific programs, some part of the code always remains

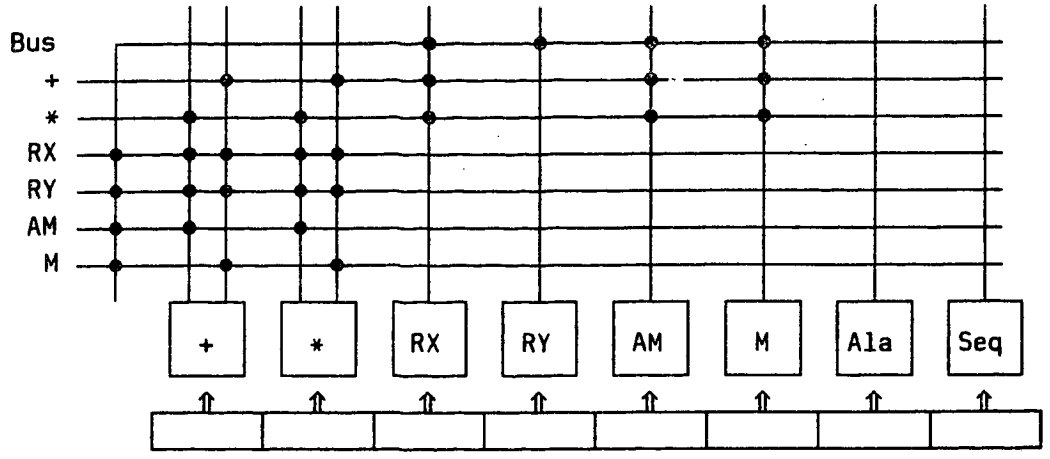
scalar. If performances in scalar mode are too bad beside vector performances, the execution time of the residual sequential code may become predominant. On Cray1, no advance can be taken at the decode sequence. For example, when the loop 1 is executed in scalar mode, about 25 cycles are necessary to decode the loop body: the theorical throughput of the memory is then more than six times higher than the real throughput!

Even in the case of a register machine where advance can be taken at the decode [To67][We84], it is very difficult to busy all the FUs in scalar mode : parallel decode is quite impossible for register machines. The decoder will always remain a bottleneck for performances in scalar mode for these machines: that is the reason why they are considered as vector machines.

3 Microcoded full connected pipelined machines

In microcoded machines, an instruction word countains a subinstruction parcel for each of its FUs. When all operands are loaded from registers, the register throughput becomes tremendous and unrealistic. Operands for a FU may be directly taken on output buses of the FUs. On a full connected machine, each output of a FU is connected to each input of the FUs; some connections are not so usefull as others: e.g. the path between the output of the floating point adder and the input of the address unit is very rarely used; such pathes may be suppressed (fig. 1).

The FPS 164 of Floating Point Systems [Ch81] may be considered as a "nearly" full connected pipeline processor. Each of its FUs can execute an instruction on every cycle. As the decoder can



A data interconnection scheme in the FPS 64
fig. 1

decode an instruction word on every cycle, the FPS 164 can run at full speed on scalar code.

In an instruction word, the subinstruction parcel for each FU is always located at the same place in the word. For example, the subinstruction for the adder can always be decomposed in three subparcels:

- Origin of the first operand, (i.e. the bus on which it is present)
- Origin of the second operand
- Codeoperation

The width of an instruction word for the FPS 164 is only 64 bits; it contains an order for ten differents FUs (see [Ch81]). Controls of the crossbar network are given by the origins of the operands for the different FUs; synchronization of the FUs is explicit: the orders decoded at a cycle are initiated at the next cycle. A datum must be present on the output bus of a FU at the

foreseen cycle. Delays to cross most of the FUs are constant and the date of their exit from a FU can be foreseen. Unfortunately, unforeseen memory banks conflicts may arise on an interleaved memory ; in this case, a solution consists in stopping the clock for the other FUs.

It is very difficult to produce efficient code for the FPS 164 because of the explicit synchronization by the microcode between the FUs. All the instructions are scalar i.e. an instruction initiates at most one order by FU. This is not critical because an instruction word can busy all the FUs; but automatic production of dense code is very difficult. Moreover efficient code cannot be produced without unrolling loops. E.g., on FPS 164, the loop body of the inner dot product may be coded on a single instruction loop preceeded by about ten instructions to fill the pipeline and about ten instructions to empty the pipeline. Unfortunately, no general compiling techniques can be used to unroll loops. To allow high performances on the FPS 164, Floating Point Systems proposes a library of frequently used functions and procedures which have been handly coded (BLAS). Performances are increased by a factor of two when coding LINPACK using BLAS instead of coding it in classical FORTRAN [Do85]. When possible hazards may occur (e.g. loop 3), loops cannot be unrolled: accesses are performed in the order of the decode.

We have pointed out the major limitations of some existent pipeline machines. Performances of existent vector register machines such as Cray1 dramatically drop out as soon as "good"

conditions disappear, (scalar code, memory banks conflicts, scatter-gather, possible hazards ..). In scalar mode, performances are limited by the decoder: at each cycle, only one instruction can be decoded and then only one FU can be activated, moreover the decoder cannot take any advance and no overtaking of the writes by the reads is allowed on memory. Full connected microcoded pipeline processors can run at full speed in scalar mode, but they present unsurmountable difficulties for efficient code production. For both models of machines, parallelism between the FUs is generated by software : possible hazards (e.g. loop 3) induce scalar execution, overlapping of the end of a loop by the beginning of the next loop is rarely performed and is always poor.

In section III, we present a new model of pipeline machine. As in FPS 164, an instruction may contain a subinstruction parcel for each FU and, as the execution of these subinstructions are totally independant, a distributed decode can be done; an other parallel decoder is also proposed. Vector and scalar instructions are available for this model of machine; a good overlapping of distinct instructions is possible whether they belong to same iteration of a loop or no and even when they do not belong to the same loop.

III The Data Synchronized Pipelined Architecture (DSPA)

1 goals

The basic problems which have lead us to define a new model of pipeline machines have been developped in the previous section.

As we have already pointed out, todays' pipeline processors must be designed to be included as elementary processors in a multiprocessor computer : MIMD and SIMD computers. Using full connected microcoded pipeline processors as elementary processors in such architectures would be unrealistic: when a datum cannot be delivered by the shared memory, the clock should be stopped for the other FUs of the elementary processor; this is unacceptable and has lead us to express a first condition that we want to be satisfied by the design of a pipeline processor:

Condition 1:

The sequencings of the different FUs of a pipeline processor have to be independant

When this condition is satisfied, a delay in the delivery of a datum for a FU will not necessarily block the other FUs.

Also, an instruction is not necessarily initiated at the cycle where its operands are produced and the productions of the operands of an operation has not to be synchronized.

In section I, we have seen that the decoder becomes a bottleneck for performances in scalar mode for vector register pipelined machines, then we enounce a second condition:

Condition 2:

Parallel decode is necessary on pipeline processors.

We have also noticed that performances on the FPS 164 depend heavily on handly coded libraries. We wish to design machines

able to reach correct performances on a very large set of scientific programs. This has lead us to a third condition:

Condition 3:

Natural code generation (by a compiler) must produce efficient code.

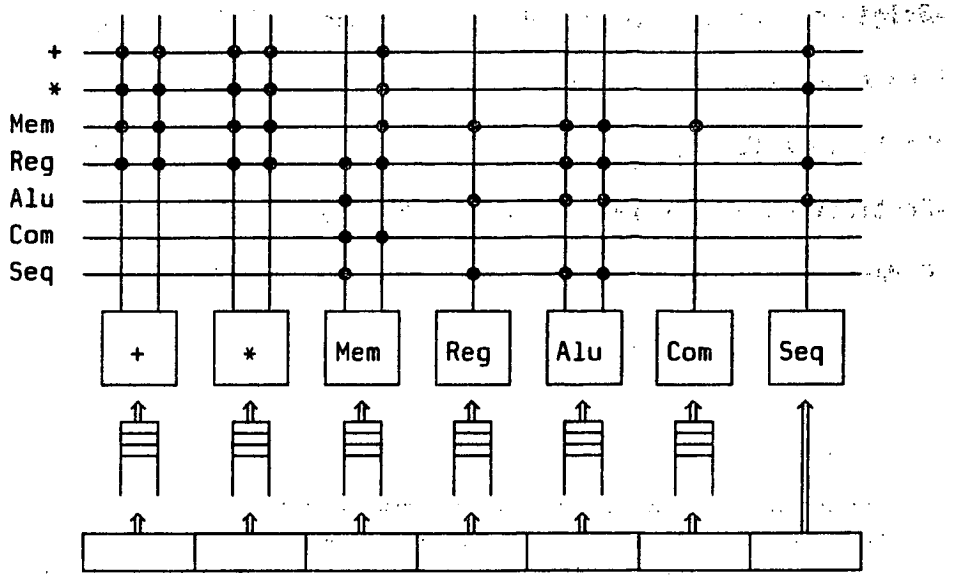
Two hardware tools seem to favor this goal: possible advance on decode and a RAW detection mechanism to enable the reads to overtake the writes on memory.

2 The DSPA model

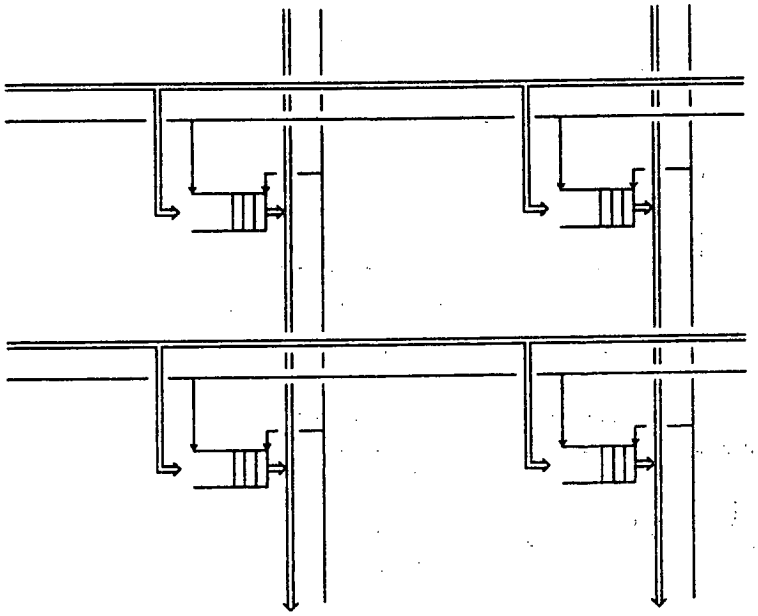
We have rejected register pipeline machines because of the tremendous throughput demanded on the registers and on the decoder to achieve performances in scalar mode. A Data Synchronized Pipeline processor is a "nearly" full connected pipeline processor in which a FIFO (First In, First Out) is associated with each crosspoint on the interconnection scheme between the outputs of the FUs (from now, an output of a FU will be refered as a producer) and the entries of the FUs (from now, an input of a FU will be refered as a consumer). Except for the sequencer, each FU has also a FIFO of instructions (fig. 2). When a datum flows out from a producer P, it is stored in a FIFO associated to a pair (P,C) where C is a consumer i.e an input of a FU F: the datum is stored in this FIFO until F is ready to treat it (fig.3).

The sequencing of the FUs is very simple :

An instruction for a FU may be decomposed in three parcels :



An example of DSPA integration scheme
fig. 2

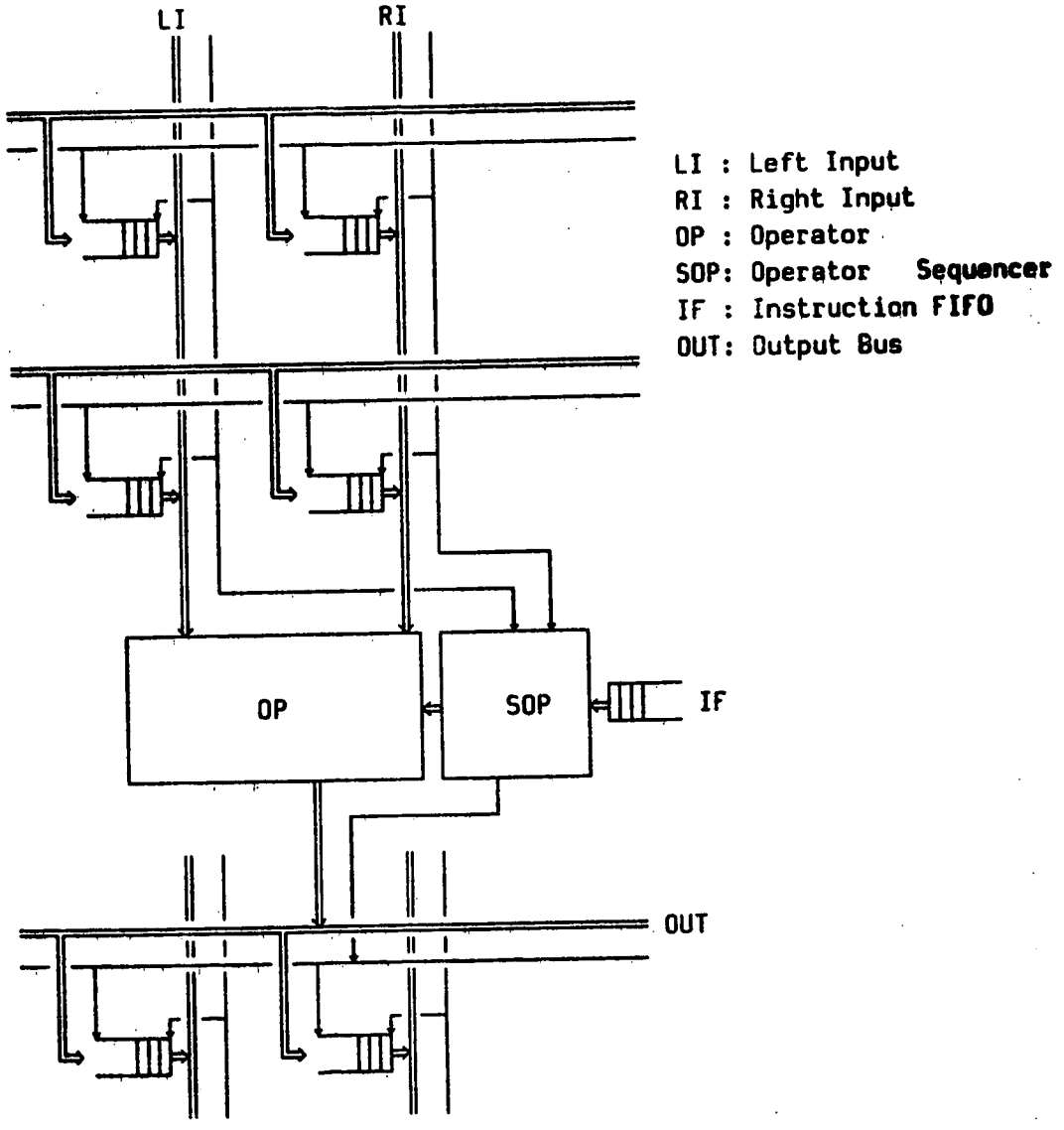


Crosspoint FIFOs
fig. 3

-Origin(s) of the operand(s) (i.e. name(s) of producer(s); the FIFO associated to the path between the producer and the input of the FU is refered)

-Destination of the result (i.e. the name of a consumer)

-Codeoperation



A typical DSPA functional unit
fig. 4

We detail here the sequencing of the adder (fig. 4)

1.If the FIFO of instructions is empty then GOTO 1.

2.Load the instruction in the adder's sequencer and decode it.

3.If one of the origin FIFOs is empty then GOTO 3.

4.Load the operands; initiate the operation.

5.Store the result in the refered FIFO and GOTO 1.

These five steps may be pipelined:

-Step 5 is always overlapped by the other steps;

-The next instruction (when existing in the FIFO) may be decoded during step3 and 4;

-Operands may always be loaded and the operation initiated: this operation can be aborted if the operands are not valid.

During the sequencing, the FIFO of instructions may accept instructions at anytime.

3 Deterministic execution

In classical machines, instructions are executed in the same order they are decoded: this guarantees the signification of a sequence of instructions. This constraint is respected on the FPS 164: instructions are immediatly executed.

In our model, the FUs are synchronized by the data; an instruction may wait for its operands during a few cycles: there is no reason to decode the memory load of an operand for an addition before decoding the addition; if the adder is not busy, it can wait for the datum. The only significant order in this

model of pipeline machines is the order of the data which enter the same FIFO.

A FIFO is associated to only one producing functional unit. The instructions FIFO of a FU guarantees that this FU initiates its operations in the same order it receives its instructions: it is reasonable to impose that the results flow out from the FU in the same order their production is initiated: in most of the cases, the delay to cross a FU is constant.

This constraint guarantees the unicity of the signification of a code sequence.

4 DSPA interleaved memory

In the DSPA model, the memory FU can be considered as a producing unit (reads) and also as a consuming unit (writes). The DSPA model imposes the results of the reads to flow out from the memory in the same order the reads have been decoded. On an interleaved memory, this may dramatically decrease the throughput of the memory if the reads are really done in the same order they are decoded:

Let us suppose a four memory bank interleaved memory which banks are busied during four cycles by a read. Let us suppose a sequence of eight consecutive read requests distributed in the following manner:

read 1 and 2 on bank 0

read 3 and 4 on bank 1

read 5 and 6 on bank 2

read 7 and 8 on bank 3

The results flow out from the banks in the good order if the reads are initiated in the good order:

read 1 is initiated on cycle 1

read 2 is initiated on cycle 5

(bank 0 is busy on cycles 2, 3, 4)

read 3 is initiated on cycle 6

read 4 is initiated on cycle 10

read 5 is initiated on cycle 11

read 6 is initiated on cycle 15

read 7 is initiated on cycle 16

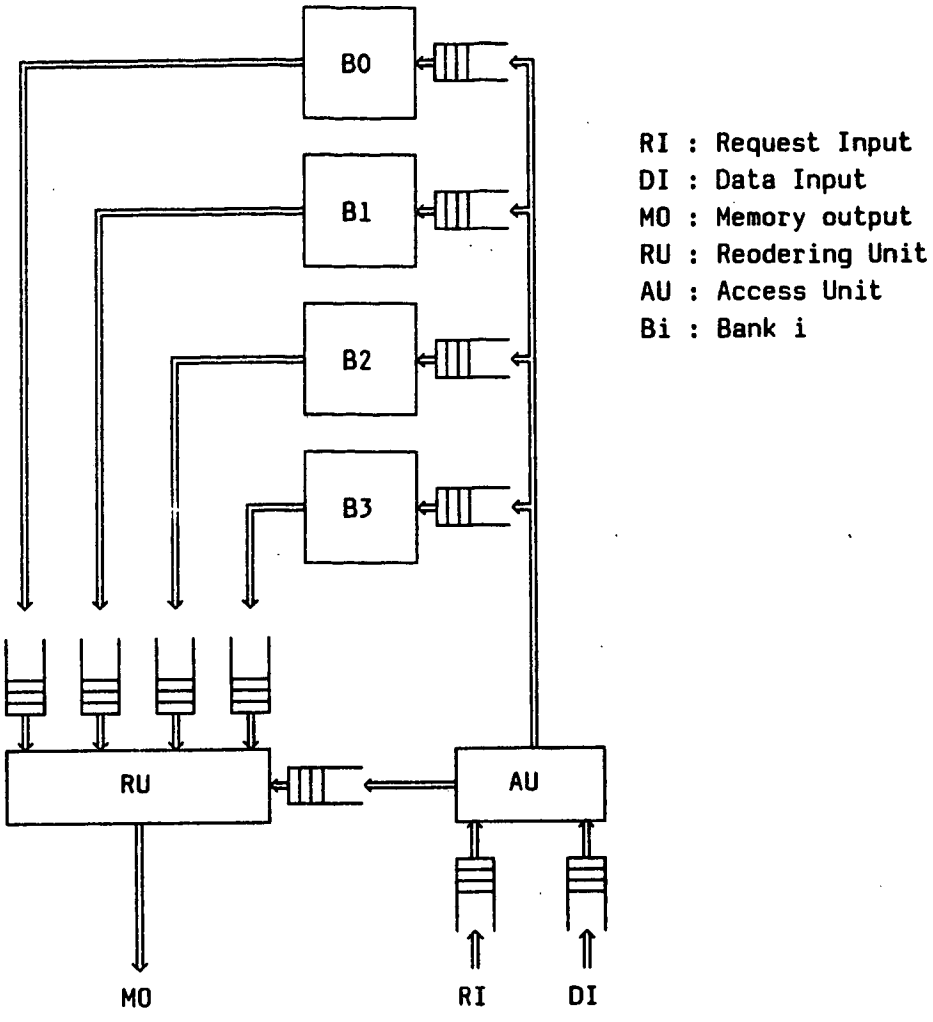
read 8 is initiated on cycle 20

In this extreme case, the real throughput of the memory is only equal to the two fifths of the theorical throughput.

We propose a design of an interleaved memory which increases the real throughput of the memory and which respects the DSPA philosophy (fig. 5).

Addresses (and data for the write instructions) flow out from the access unit (AU) in the good order and enter the FIFO of requests of the desired banks. Bank i loads its requests from the FIFO, read data are stored in a FIFO associated to bank i. Then the reordering unit (RU) reads the data on the desired FIFOs. The data flows out from the memory unit on the memory output bus (MO) in the desired order.

One can easily verify that when the sequence of reads of the previous example is repeated, the asymptotic throughput of the memory reaches the theorical throughput.



A DSPA-compatible interleaved memory
fig. 5

We will see later that a RAW detection hardware mechanism is necessary on the memory of a pipeline machine; this mechanism can be implemented in the AU (global detection) as well as in each bank (local detection).

5 Instruction decode

5.1 A distributed decode

We have already pointed out that the relative order of two instructions for two distinct FUs does not matter.

Many solutions may be imagined for the global decoder.

For example, as in the FPS 164, an instruction may contain a subinstruction parcel for the distinct FUs: these subinstructions may always be located at the same places in the instruction word and then can be directly routed to the distinct instructions FIFOs of the FUs:

The global decoder is only a bus.

When, in an arbitrary sequence of instructions (e.g. the body of loop), the most frequently used FU receives N instructions, the whole sequence can be coded on only N instruction words; the relative order of the subinstructions for the same FU is the only significant relations.

Very dense code is obtained without unrolling the loops as for the FPS 164 for example. Moreover, in the FPS 164 case, a lot of instruction words are necessary to fill and to empty the pipeline : in our model, these instructions are not necesssary; all the iterations of a loop have the same code. Let us suppose a DSPA processor which FUs have the same characteristics as in the FPS 164. Natural code generation will produce only one instruction by FU for the loop body of the inner dot product: without unrolling loops, a compiler will generate a single

instruction word loop for this machine.

Nevertheless, we do not think that this solution of the decode is the only possible : 64 bits are necessary to code an instruction word for the FPS 164, more informations must be given in subinstructions in our model, the width of an instruction word may induce a too expensive instruction memory or cache.

5.2 Decoder throughput

We expose here why the throughput of the decoder may be more limited.

5.2.1 Flexibility of the model

In the design of the FPS 164 and other machine as Cray1, all the FUs are assumed to have the same basic cycle : this allows to initiate a new instruction on each FU at each cycle. This simplifies the microcoding of the FPS 164 and the sequencing of the Cray1.

But this cannot be considered as a good balance between the throughput of the FUs: most of the classical algorithms require at the minimum an average of one memory access per floating point operation. On the FPS 164, this difficulty has lead to the introduction of an auxiliary memory having the same throughput as the main memory. In the design of the successor of the Cray1, the Cray-XMP, two pipelined channels are used to access simultaneously the memory: they can be considered as distinct FUs which have to respect very strong constraints.

Our model of pipeline processor can support distinct basic cycles for the distinct FUs; e.g. the basic cycles of the

floating point operators may be two times longer than the basic cycle of the memory. Moreover the delay in which a FU delivers a result may vary; we have only imposed the results to flows out from the FU in the order their productions have been initiated.

5.2.2 Vector instructions

On the FPS 164, vector instructions cannot exist because of the explicit synchronizations by the microcode at each cycle. In the DSPA model, no difficulties exist to introduce vector instructions. Executing a vector instruction of length k consists in executing the same scalar instruction k times. In some cases as in loop 4, vector instructions increase the performances by removing strict dependances between the data in the pipeline :

Example 4:

```
      Do 4 I=1,N
    4 A(I)= B(I) + C(I) + D(I)
```

In the body of this loop, two instructions concern the adder; the second addition cannot be initiated before the end of the first one. The number k of cycles to cross the adder may be high (seven cycles in Crayl); no additions can be initiated during k-1 cycles. A vector instruction of length k works on k independant flows of data : in loop 4, the adder should then be busied when the memory throughput is sufficient. Our model allows to code many loops as vector loops; loops 1 and 2 are coded with vector instructions; the following loop 5 is also coded as a vector loop.

Example 5:

```
      DO 5 I=1,N
    5 A(H(I))= B(P(I))*C(Q(I))
```

The loop 3 may be coded by mixing vector instructions and scalar instructions; accesses to H, B, C, the multiplication and also the addition may be coded as vector instructions. In order to detect possible RAW hazards, the two accesses to A must be scalar coded: an internal scalar loop has to be coded in the global loop.

Let us suppose that the maximum vector length of a vector instruction is 8, the body of loop 3 can be coded by the following sequence :

```
V: pic(1) Vector Read H --> memory address input
   pic Vector Read B --> left input of the multiplier
   pic Vector Read C --> rigth input of the multiplier
   Vector * (from memory, from memory) --> left input of the adder
   Vector + (from multiplier, from memory) -->
                              data input of the memory
```

---

(1) pic : postincrement; this instruction refers a descriptor containing a base address A and an increment R. A vector pic generates N addresses of the form A + iR and leaves the descriptor with a new base address A + NR; a scalar pic sends the address A to the memory and assigns the value A + R to A.

```
S:  Indirect Read A ( address coming from memory ) -->
                                        rigth input of the adder
    pic Store A from adder
sequencer:  C<-- C-1, if C>0 then JUMP to S
sequencer:  N<-- N-8, if N>0 then
                    ( VL<-- min(N,8); C<-- VL; JUMP to V).
```

We suppose that the memory FU computes postincremented addresses and indirect based addresses. The floating point additions have been extracted from the internal scalar loop : these instructions are kept in the adder's instruction FIFO until the operands are received.

The flexibility of the vector instruction definition associated with the possibility to access vector operands in scalar mode and to concatenate scalar operands to form vector operands increases the ratio of vector instructions in DSPA programming.

5.2.3 Towards an other proposition for distributed decode

In a loop body of scientific programs, some FUs are not so used as other; in the previous examples, no registers were used. When an instruction word may contain a subinstruction parcel for each FU, the ratio of noop instructions parcels in a sequence of code may be tremendous. E.g., in most of the cases, there is only one instruction for the sequencer in a loop body; on the other hand, in all classical examples the memory is the critical

ressource : it does not seem natural to decode an instruction for the sequencer on every cycle, but it may be critical for the memory. Our experiments have shown that for a machine with well designed FUs, nearly half of the instructions are memory accessses. We recall that we consider that the memory FU computes postincremented addresses and indirect based addresses.

Then we have imagined to decode only a few instructions at the same cycle. These instructions must be applied to distinct FUs. It seems that decoding two instructions during a memory cycle represents a good balance between the decoder throughput and the possible performances of the machine. The global decode is very simple : names of the FUs involved by the instructions are decoded and then the two instructions are routed to the correct instructions FIFOs.

IV RAW hazards

The DSPA model respects the conditions 1 and 2. When producing code for a processor of DSPA family, the compiler can forget that this processor may be an elementary processor of a multiprocessor computer. The code may be generated in the same manner it would have been generated for a monoprocessor: the real behavior of the shared memory can be ignored. This has allowed to treat loop 2 and 5 as vector loops: this cannot be done on the Cray1 because the behavior of the memory cannot be foreseen at compile time.

Independancy of the FUs also allows parallel decode (Two

solutions have been proposed, others may be imagined). On the other hand, code generation may be relatively simple : as the instruction flows for two distinct FUs are completely independant, the compaction of a linear sequence of code is very easy.

Unfortunately advance on decode is not very useless when the effective accesses on memory have to be done in the order of their decode; in most of the cases, a loop body begins by a read and ends by a write on memory. This forbids the overlapping of the end of an iteration of the loop by the beginning of the next iteration unless some mechanism is used to allow passing the writes by the reads (when the read address and the write address are distinct). Software mechanisms have been proposed in the pass: two distinct flows of memory accesses may be generated and reads of a flow overtake the writes of the other flow. Many examples where these solutions are not efficient can be imagined.

Example 6:

```
      DO 6 I=1,N
      DO 10 J=1,I
   10 A(J)=A(J)+B(I,J)
    6 CONTINUE
```

When executing the internal loop, it is very interesting to pass the write of A(J) by the reads of A(J+1), A(J+2),.. . But when I becomes small, one cannot ensure that A(1) has been written by the previous external iteration. The only software solution to prevent hazards in the execution of this loop is to empty the pipeline after each iteration of the internal loop.

We think that a hardware detection of RAW hazards on memory has to be implemented in a pipeline processor; in loop 3, there are possible hazards on memory, no software means can be imagined to treat this case : without hardware detection of hazards this loop must be treated in scalar mode (at least the writes and reads on vector A must be done in the order of the decode). Using one of the two models of decode we have presented, the decode of this loop programmed with vector instructions and an internal scalar loop will take no more than 20 cycles: the decode will not be a bottleneck (40 accesses to the memory are done). If a hardware detection of RAW hazards is done, one can hope that when real hazards don't occur, performances will only be limited by the memory throughput.

Another advantage of a hardware detection of hazards is to enable the reaching of correct performances on unoptimized code : Performances on loop 1 when scalar coded can be equal to vector performances, even where start-up delays are longer if an interleaved memory is used. One can hope correct performances on long loop bodies.

V Some limitations of the DSPA model

1 Registers·

In the examples we have detailed, the registers have never been used. In the case where a datum is used two times or more in an algorithm, it must be explicitly saved in a register and each operation requiring this datum as an operand will cost two

instructions: the instruction to initiate the operation and the read of the register. The most important case of two uses of the same operand is the complex multiplication: a solution may consist in implementing a complex mode on the multiplier.

2 A blocking machine

In a DSPA processor, an instruction waits for its operands; this instruction may have been decoded before the decode of the production of its operands and if these productions are never done (a bad generation of code may lead to this extremity), the FU will never be activated as in example 7:

7: + (from memory, from memory)--> Left input of the adder

Sequencer: JUMP to 7

On the other hand, data may be produced and never consumed:

Example 8:

8: Pic Read X --> Left input of the adder

Sequencer: JUMP to 8

Correct codes must be written: when a datum is produced by a FU, it must be consumed by an other; on the other hand when a data has to be consumed, its production has to be guaranteed.

We define a correct unbreakable sequence of code (CUS) as:

No external jump inside the sequence is possible unless to its first instruction.

No jump out of the sequence is possible unless from the last instruction.

Each datum produced in the sequence is consumed in the sequence.

Each datum consumed in the sequence is produced in the sequence.

E.g., the loop body of example 3 is a CUS, but the internal scalar loop is not a CUS: Data which are consumed are not produced in the loop.

The following loop body of an inner dot product is not a CUS:

S: Pic Load A --> Left input of the multiplier

Pic Load B --> right input of the multiplier

* (from memory, from memory) --> right input of the adder

+ (from adder, from multiplier) -->

left input of the adder

sequencer: C<-- C-1, if C>0 then JUMP to S

When executing this loop, the first left operand for the adder must have been previously produced in order to initiate the pipeline; this corresponds to the initialization of the sum.

The following condition must be respected to guarantee the execution of all the decoded instructions:

Each sequence of code must be included in a CUS.

There are no problems for a compiler to produce correct code (e.g. a solution may consist in treating DO loops as CUS, addresses of jumps delimiting distinct CUSs). Hand optimizations of the code may be dangerous: if the previous condition is not respected, the machine will be blocked. But this is not a real problem:

machines on which bad coded programs produce the desired results don't really exist.

VI Conclusion

We have presented an original model of architecture for pipeline processors. In this model, the decoder does not remain a bottleneck for performances in scalar mode as in existent pipeline processors : independancy of the FUs and distributed decode of their instructions allow a very fast sequencing. Synchronization of the FUs is done by the data. Such a processor can be integrated in synchronous multiprocessor architectures as well as in classical MIMD structures.

Generating code for machines based on our model will be very easy; natural sequential code can first be generated (instruction by instruction), then independant codes for the distinct FUs can be extracted. For classical pipeline computers, complex reordering algorithms are applied on code at compile time to ensure performances at execution time, these algorithms may also be applied on the distinct codes for the FUs but this is less necessary than for classical architectures. When possible hazards may occur on memory or registers, there is no mean to optimize code for classical pipeline computers; On a DSPA architecture only real hazards may degrade performances.

At present, we are studying a real implementation of a pipeline processor of this family. Many solutions can be adopted: buses can be shared by several FUs, this share may be static or dynamic ... A great attention has to be taken in the design of the FUs; for example, in this paper we have already exposed some characteristics of the memory FU (computation of the addresses by

the FUs, need of a RAW hazard detection mechanism). Another important point to be studied in the design of the machine is the interconnection scheme : connections can be quite expensive in this mode because a FIFO is associated to each crosspoint on this interconnection scheme; unusually used paths between the FUs can be suppressed.

In the theorical model, arbitrarily large FIFOs are used. In a real machine, the sizes of the FIFOs are limited by the physical support and several FIFOs may be implemented on the same support. All these questions will be discussed in future papers.

Bibliography

[Ba80] J.L.Baer, *Computer Systems Architecture,* Computer Science Press, 1980

[Ch81] A.E.Charlesworth, "An approach to scientific array processing: the architectural design of the AP120B/FPS 164 Family" Computer, september 1981

[Cr79] Cray-1 Computer Systems, *Hardware Reference Manual,* Cray Research Inc., Chippewa Falls, WI 1979

[Do85] J.J.Dongarra, "Performances of various computers using standard linear equations software in a FORTRAN environment", Computer Architecture News, pp3-11, march 1985

[Ga83] D.Gajski, D.Kuck, D.Lawrie, A.Sameh, "Cedar: a large scale multiprocessor", International Conference on Parallel Processing 1983, pp524-529

[Go83] A.Gottlieb & al., "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer" IEEE Transactions on Computers, Vol. C-32, pp175-189, feb.1983

[Ho81] R.W.Hockney, C.R.Jesshope, *Parallel computers: architecture, programming and algorithms,* Adams Hilger, Bristol 1981

[Ho84] R.W.Hockney, "MIMD computing in the USA - 1984", Parallel Computing, 1985, pp119-136

[Hw84] K.Hwang, F.A.Briggs, *Computer architecture and parallel processing,* Mac Graw Hill 1984

[Ko81] P.M.Kogge, *The architecture of pipelined processors,* Mac Graw Hill 1981

[Ku82] D.J.Kuck, R.A.Stokes, "The Burroughs Scientific Processor (BSP)", IEEE Transactions on Computers, vol C-31, pp. 363-376, May 1982.

[Kung79] H.T.Kung, "The structure of parallel algorithms", Dept of Computer Science CMU, Aout 1979

[La75] D.H.Lawrie,"Access and alignment of data in an array computer", IEEE Transactions on Computers, vol C-24, pp.1145-1155, dec.1975.

[La82] D.H.Lawrie, C.R.Vora, "The prime memory system for array access", IEEE transactions on Computers, vol C-31, pp. 435-442, May 1982.

[Ra77] C.V.Ramamoorthy, H.F.Li, "Pipeline Architecture", Computing Surveys, Mars 1977

[Si81] H.J.Siegel & al., "PASM : a partitionable SIMD/MIMD system for image processing and pattern recognition", IEEE Transactions on Computers, Vol C-30, pp934-947, 1981

[Sm78] B.J.Smith, "A pipelined shared resource MIMD computer ", IEEE Proceedings 1978 International Conference on Parallel Processing, pp6-8

[Ta85] H.Tamura, Y.Shinkai, F.Isobe, "The supercomputer FACOM VP system", Fujitsu Sc. Tech. J. March 1985

[To67] R.M.Tomasulo, "An efficient algorithm for exploiting multiple arihmetic units", IBM J., Vol. 11, Jan. 1967

[We84] S.Weiss, J.E.Smith, "Instructions issue logic in pipelined supercomputers", Transactions on Computers, pp1013-1022, Nov. 1984

PI 278 **Controlling knowledge transfers in distributed algorithms – Application to deadlock detection**
Jean – Michel Hélary, Aomar Maddi, Michel Raynal – 32 pages ; Janvier 86.

PI 279 **Une méthode de conception de programmes fonctionnels**
Raymond Durand, Martine Vergne – 16 pages ; Janvier 86.

PI 280 **Commande de systèmes redondants et évitement d'obstacles**
Bernard Espiau – 52 pages ; Janvier 86.

PI 281 **A distributed algorithm for mutual exclusion in an arbitrary network**
Jean – Michel Hélary, Noël Plouzeau, Michel Raynal – 16 pages ; Janvier 86.

PI 282 **Stabilité robuste dans la commande adaptative indirecte passive**
Philippe de Larminat – 70 pages ; Janvier 86.

PI 283 **Data synchronized pipeline architecture pipelining in multiprocessor environments**
Yvon Jégou, André Seznec – 36 pages ; Janvier 86.

Publication interne
n° 283

Janvier 1986

Yvon Jégou

André Seznec

DATA SYNCHRONIZED PIPELINE ARCHITECTURE PIPELINING IN MULTIPROCESSOR ENVIRONMENTS

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique